La Blockchain

Introduzione

I dati della struttura dati blockchain sono un'ordinata lista di blocchi di transazioni back-linked (ndt ogni blocco è collegato al blocco precedente). La blockchain può essere salvata come un file piatto, o in un semplice database. Il client Bitcoin Core salva i metadati della blockchain usando il database LevelDB (libreria C scritta dai programmatori che lavorano a Google). I blocchi sono collegati "indietro", ognuno si riferisce al blocco precedente presente nella catena. La blockchain è spesso visualizzata come una pila verticale, con blocchi stratificati l'uno sopra l'altro e il primo blocco serve da fondamenta della pila. La visualizzazione dei blocchi impilati uno sopra l'altro provoca l'uso di terminologie come "altezza" (height) per riferirsi alla distanza dal primo blocco, e "top" o "tip" (cima o punta) per riferirsi al blocco aggiunto più recentemente.

Ogni blocco contenuto nella blockchain è identificato da un hash, generato usando l'algoritmo crittografico di hashing SHA256 sull'header del blocco. Ogni blocco inoltre referenzia il blocco precedente, conosciuto anche come *parent* block, attraverso il campo "previous block hash" nel block header. La sequenza di hash che collegano ogni blocco al proprio parent crea una catena che si collega, blocco per blocco fino al primo blocco creato, conosciuto con il nome di *genesis block* , ovvero blocco di genesi.

Anche se un blocco ha solo un genitore, può temporaneamente avere multipli figli. Ognuno dei figli si riferisce allo stesso blocco del padre e contiene lo stesso hash (padre) nel campo "previous block hash" (hash del blocco precedente). Multipli figli emergono durante un "fork" (biforcazione) della blockchain, una situazione temporanea che accade quando differenti blocchi sono scoperti quasi simultaneamente da miner differenti (vedi [forks]). Eventualmente, solo un blocco figlio diventa parte della blockchain e la "biforcazione" viene risolta. Anche se un blocco può avere più di un figlo, ogni blocco può avere solo un genitore. Questo perchè un blocco ha un singolo campo "hash del blocco precedente" (previous block hash) che si riferisce al suo singolo genitore.

Il campo "previous block hash" è dentro l'header del blocco e per questo influenza l'hash del blocco *attuale*. L'identità stessa del figlio cambia nel caso in cui cambi l'identità del genitore. Quando il genitore viene modificato in qualsiasi modo, l'hash del genitore cambia. L'hash modificato del genitore necessita un cambio nel puntatore "previous block hash" del figlio. Questo a sua volta causa il cambio dell'hash del figlio, che richiede il cambio nel puntatore dell'hash nipote, che a sua volta cambia il puntamento al nipote, e così via... Questo effetto a cascata assicura che fino a quando un blocco ha tante generazioni di blocchi che lo seguono, non può essere modificato senza forzare un ricalcolo su tutti i blocchi seguenti. Visto che per questo ricalcolo servirebbe un'enorme potenza computazionale, l'esistenza di una lunga catena di blocchi fa si che la storia più profonda della blockchain sia immutabile, che è una degli elementi chiave della sicurezza di bitcoin.

Un modo di pensare alla blockchain è come ai livelli di una stratificazione geologica, o a una carota di ghiaccio. Gli strati superficiali possono cambiare con le stagioni o anche essere eliminati prima che abbiano il tempo di depositarsi, ma se si scende di pochi centimetri gli strati geologici diventano sempre più stabili. Al punto che arrivi a guardare poche centinaia di metri sotto stai vedendo una foto del passato che è stata lasciata intatta da milioni di anni. Nella blockchain i blocchi più recenti potrebbero cambiare se c'è un fork. Gli ultimi sei blocchi sono come alcuni

centimentri del suolo. Ma se vai più in profondità nella blockchain, oltre sei blocchi, i blocchi hanno sempre meno probabilità di essere modifcati. Dopo 100 blocchi c'è così tanta solidita che la transazione di coinbase - una transazione contenente i nuovi bitcoin minati - può essere spesa. Alcune migliaia di blocchi indietro (un mese), e la blockchain è saldata nella storia, per qualunque tipo di dato. Mentre il protocollo permette sempre che una catena possa essere sostituita da una catena più lunga ed esiste sempre la possibilità che una catena diventi reversibile, la possibilità che questo accada diminuisce coi blocchi fino a diventare infinitesima.

Struttura del Blocco

Un blocco è un contenitore di una struttura dati che riunisce le transazioni da includere nel pubblico registro, la blockchain. Il blocco è composto da un'intestazione, contenente i metadati, seguito da una lunga lista di transazioni che costituiscono la maggior parte delle sue dimensioni. L'intestazione del blocco è di 80 bytes, mentre la media delle transazioni è di almeno 250 bytes e in media un blocco contiene più di 500 transazioni. Un blocco completo, con tutte le transazioni, è perciò 1000 volte più grande di un'intestazione del blocco.La struttura di un blocco descrive la struttura di un blocco.

Table 1. La struttura di un blocco

Dimensione	Campo	Descrizione
4 byte	Dimensione del Blocco (Block Size)	La dimensione del blocco, in byte
80 bytes	Header del Blocco (Block Header)	Multipli campi dall'header del blocco
1-9 byte (VarInt)	Contatore di transazione (Transaction Counter)	Quante transazioni seguono
Variabile	Transazioni	Le transazioni registrate nel blocco

Header del Blocco

L'intestazione del blocco consiste in tre gruppi di metadata. Nel primo, c'è un riferimento al precedente hash, il quale connette questo blocco al precedente blocco nella blockchain. Il secondo set di metadata, cioè il *difficulty*,*timestamp*,e_nonce_,riguarda la competizione del mining, come dettagliato in [ch8]. Il terzo pezzo di metadata è il merkle tree root, una struttura dati usata per riassumere efficientemente tutte le transazioni nel blocco. La struttura di un block header descrive la struttura di un'intestazione del blocco (block header).

Table 2. La struttura di un block header

Dimensione	Campo	Descrizione
4 byte	Versione	Un numero di versione per tracciare upgrade al software e/o al protocollo
32 byte	Hash del Blocco Precedente	Un riferimento all'hash del blocco precedente (genitore) nella chain

Dimensione	Campo	Descrizione
32 byte	Merkle Root	Un'hash della radice del merkle tree delle transazioni di questo blocco
4 byte	Timestamp	Il tempo approssimato della creazione del blocco corrente (secondi dalla Unix Epoch)
4 bytes	Target di Difficoltà (Difficulty Target)	Il target di difficoltà dell'algoritmo di proof-of-work per questo blocco
4 byte	Nonce	Un contatore utilizzato per l'algoritmo di proof-of-work

Il nonce, il target di difficoltà, e il timestamp sono usati nel processo di mining e saranno discussi in maggiore dettaglio nel [ch8].

Identificatori di blocco: Hash del Block Header e Altezza del Blocco (Block Height)

Il primo identificatore di un blocco è il suo hash criptografico, un’impronta digitale, creata eseguendo due volte l’hash di un block header attraverso l’algoritmo SHA256. Il risultato di 32-byte di un hash è chiamato il block hask ma più specificatamente il block header hash ,<phrase role="keep-together">perchè solo il block header è usato per calcolarlo. Per esempio,</phrase> 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f è il block hash del primo blocco bitcoin che sia stato mai creato. Il block hash identifica un blocco unico e senza ambiguità e può essere indipendentemente derivato da ogni nodo da un semplice hashing del block header.

Nota che il block hash non è attualmente incluso all'interno della struttura dati dei blocchi, nè quando il blocco è trasmesso sul network nè quando è memorizzato sullo storage persistente di un nodo come parte della blockchain. invece, l'hash dei blocchi è computato da ogni nodo che il blocco ha ricevuto dal network. Il block hash viene memorizzato in un separato database come una parte dei metadata dei blocchi, per facilitare l'indicizzazione e il più rapido recupero dei blocchi dal disco.

Una seconda via di identificazione di un blocco è dalla sua posizione nella blockchain, chiamata <phrase role="keep-together"><emphasis>block height</emphasis>. Il primo blocco che sia mai stato creato ha un altezza di blocco 0 (zero) ed è la </phrase> <phrase role="keep-together"> lo stesso blocco sarà precedentemente preso di riferimento dal seguente block hash</phrase> 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. Un blocco può così essere identificato in due modi: da un riferimento del block hash o dal riferimento dell’altezza di blocco. Ogni susseguente blocco si aggiunge in cima al primo blocco che è in una posizione più alta

nella blockchain, come scatole impilate una sopra l’altra. L’altezza del blocco di Gennaio 2014 era approssimativamente 278,000 e significa che ci sono stati 278,000 blocchi accatastati in cima al primo blocco creato a Gennaio 2009.

A differenza del block hash, l'altezza del blocco non è un identificatore unico. Nonostante un singolo blocco avrà sempre una specifica ed invariabile altezza di blocco, l'inverso non è reale - l'altezza del blocco non è sempre identificata da un singolo blocco. Due o più blocchi possono avere la stessa altezza di blocco, competendo per la stessa posizione nella blockchain. Questo scenario è dibattuto nei dettagli della sezione [forks]. L'altezza del blocco non è anche una parte della struttura dati del blocco; non è memorizzata all'interno del blocco. Ogni nodo dinamicamente identifica una posizione di blocco (altezza) nella blockchain quando è ricevuto dal network bitcoin. L'altezza del blocco può essere anche memorizzata come metadata in un database indicizzato per rapidi recuperi.

TIP	Il *block hash* di un blocco identifica sempre univocamente un singolo blocco. Un blocco ha inoltre sempre una *block height* specifica. Però, non è sempre il caso che una block height specifica identifichi un singolo blocco. Al contrario, due o più blocchi possono competere per una singola posizione nella blockchain.

Il Genesis Block

Il primo blocco nella blockchain è chiamato genesis block ed è stato creato nel 2009. Esso è l'antenato comune a tutti i blocchi della blockchain, questo significa che se inizi a seguire ogni blocco e la chain all'indietro nel tempo, probabilmente risalirai al genesis block.

Ogni nodo inizia sempre con una blockchain di almeno un blocco perchè il genesis block è staticamente codificato all'interno del client software bitcoin, in modo che non possa essere alterato. Ogni nodo "conosce" sempre l'hash del genesis block e la struttura, il tempo in cui è stato creato e almeno la singola transazione. Quindi, ogni nodo ha il punto iniziale della blockchain, una sicura "radice" da cui costruire una blockchain affidabile.

Visualizza il genesis block staticamente encoded nel client Bitcoin Core, in chainparams.cpp.

Il seguente hash identificativo appartiene al genesis block:

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Puoi cercare quel block hash in un qualsiasi sito block explorer, come blockchain.info, e troverai una pagina che descrive il contenuto di questo blocco, con un'URL contenente quell'hash:

https://blockchain.info/block/
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

https://blockexplorer.com/block/
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Usando il client di riferimento Bitcoin Core da riga di comando:

```
$ bitcoind getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

```
{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 308321,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "nextblockhash" :
"00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

Il genesis block contiene un messaggio nascosto in esso. L'input della transazione di coinbase contiene il testo " Dal Times del 03 gennaio 2009, Il Cancelliere sull'orlo del secondo bailout per le banche" Questo messaggio serviva per provare la prima data del blocco che era stato creato, riferendosi al quotidiano britannico *Il Times*. Serve anche come un promemoria ironico dell'importanza di un sistema monetario indipendente, con il lancio di bitcoin avvenuto contemporaneamente ad una crisi monetaria mondiale senza precedenti. Il messaggio è stato incorporato nel primo blocco di Satoshi Nakamoto, creatore di bitcoin.

Collegando i Blocchi nella Blockchain

Bitcoin full nodes conserva una copia locale della blockchain, partendo dal genesis block. La copia locale della blockchain è costantemente aggiornata ai nuovi blocchi che vengono trovati e usati per estendere la chain. Quando un nodo invia ai blocchi dal network, esso validerà questi blocchi e dopo li collegherà alla blockchain esistente. Per stabilire un collegamento, un nodo esaminerà l'inizio del block header e guarderà al precedente block hash.

Assumiamo, per esempio, che il nodo abbia 277,314 blocchi nella copia locale della blockchain. L'ultimo blocco del quale il nodo sa qualcosa è il blocco 277,314, con un hash dell'header del blocco di 00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249

Il nodo bitcoin in seguito riceve un nuovo blocco dalla rete, che lo analizza come riportato qui:

```
{
    "size" : 43560,
    "version" : 2,
    "previousblockhash" :
        "0000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
    "merkleroot" :
        "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
    "time" : 1388185038,
    "difficulty" : 1180923195.25802612,
    "nonce" : 4215469401,
    "tx" : [
        "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

 #[... molte altre transazioni omesse ...]

        "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
    ]
}
```

Osservando questo nuovo blocco, possiamo notare che il nodo trova il campo previousblockhash, che contiene l'hash del blocco genitore. E' un hash conosciuto al nodo, come l'ultimo blocco della chain all'altezza (block height) 277,314. Quindi questo nuovo blocco è un figlio dell'ultimo blocco della chain e estende la blockchain esistente. Il nodo aggiunge questo nuovo blocco alla fine della catena (chain), rendendo la blockchain più lunga con una nuova height di 277,315. Blocchi collegati in una catena, per referenza al precedente hash dell'header del blocco mostra la chain di tre blocchi, collegati da referenze nel campo previousblockhash.

I Merkle Tree

Ogni blocco della blockchain di bitcoin contiene un sommario di tutte le transazione nel blocco, usando un *merkle tree*.

Un *merkle tree*, conosciuto anche come un *binary hash tree*, è una struttura dati usata per indicizzare efficientemente e verificare l'integrità di un grande gruppo di dati. Merkle trees sono binari trees contenenti gli hashes crittografici. Il termone "tree" è usato nella scienza informatica per descrivere una ramificazione della struttura dati, ma questi trees sono visualizzati solitamente sottosopra con la "radice" in alto e il "leaves" in fondo al diagramma, come vedrete negli esempi che seguono.

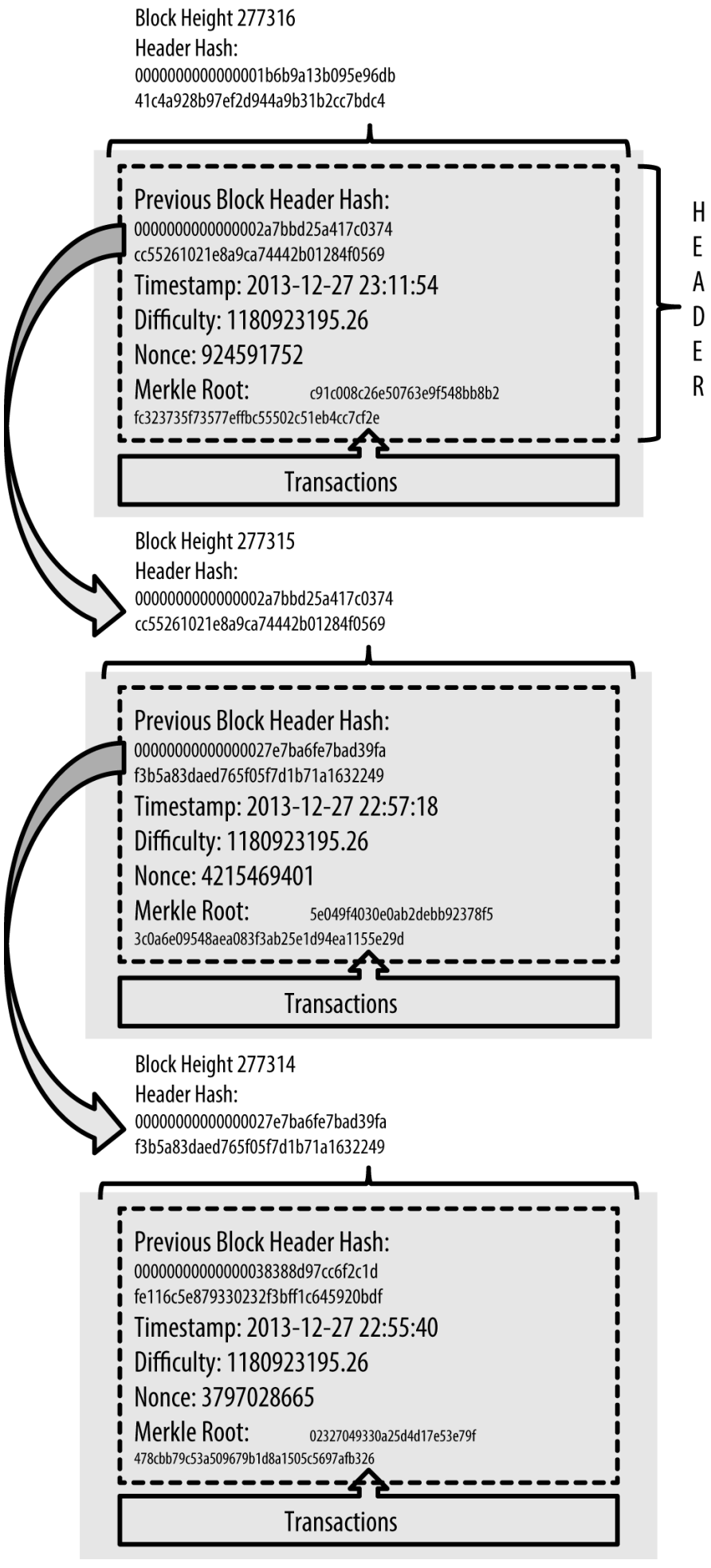

Figure 1. Blocchi collegati in una catena, per referenza al precedente hash dell'header del blocco

I merkle tree sono usati in bitcoin per indicizzare tutte le transazioni in un blocco, producendo in sostanza un'impronta digitale dell'intero set di transazioni,provvedendo ad un efficientissimo processo di verifica se una transazione è inclusa nel blocco. Un Il Merkle tree è costruito dal continuo hashing delle coppie di nodi finchè c'è un solo hash, chiamato il *root* oppure *merkle root*. L'hash dell'algoritmo crittografico usato nel merkle tree di bitcoin è SHA256 applicato due volte, conosciuto anche come doppio-SHA256.

Quando N elementi di dati sono hashati e indicizzati nel merke tree, si possono verificare se ogni singolo data element è incluso nel tree con al massimo 2*log~2~(N) calcoli, ottenendo così un efficientissima struttura dati.

Il merkle tree è costruito dal basso verso l'alto. Nel seguente esempio, partiamo con quattro transazioni, A, B, C e D, che formano le *foglie* del Merkle tree, come mostrato in Calcolando i nodi in un merkle tree. Le transazioni non sono salvate nel merkle tree; invece, i loro dati sono hash-ati e l'hash risultante è salvato in ogni nodo figlia come H_A, H_B, H_C, e H_D:

```
H~A~ = SHA256(SHA256(Transaction A))
```

Le consecutive coppie di nodi leaf sono poi indicizzate in un nodo padre che concatena i due hash e li hasha insieme. Per esempio, per costruire un nodo padre H_{AB}, i due 32-byte degli hash dei bambini sono concatenati per creare una stringa di 64-bytes. Questa stringa è poi doppiamente hashata per produrre l'hash del nodo padre:

```
H~AB~ = SHA256(SHA256(H~A~ + H~B~))
```

Il processo continua finchè c'è solo un nodo in cima, il nodo chiamato Merkle root. Questo hash da 32-byte è memorzzato in un block header e indicizza tutti i dati in tutte di tutte le quattro transazioni.

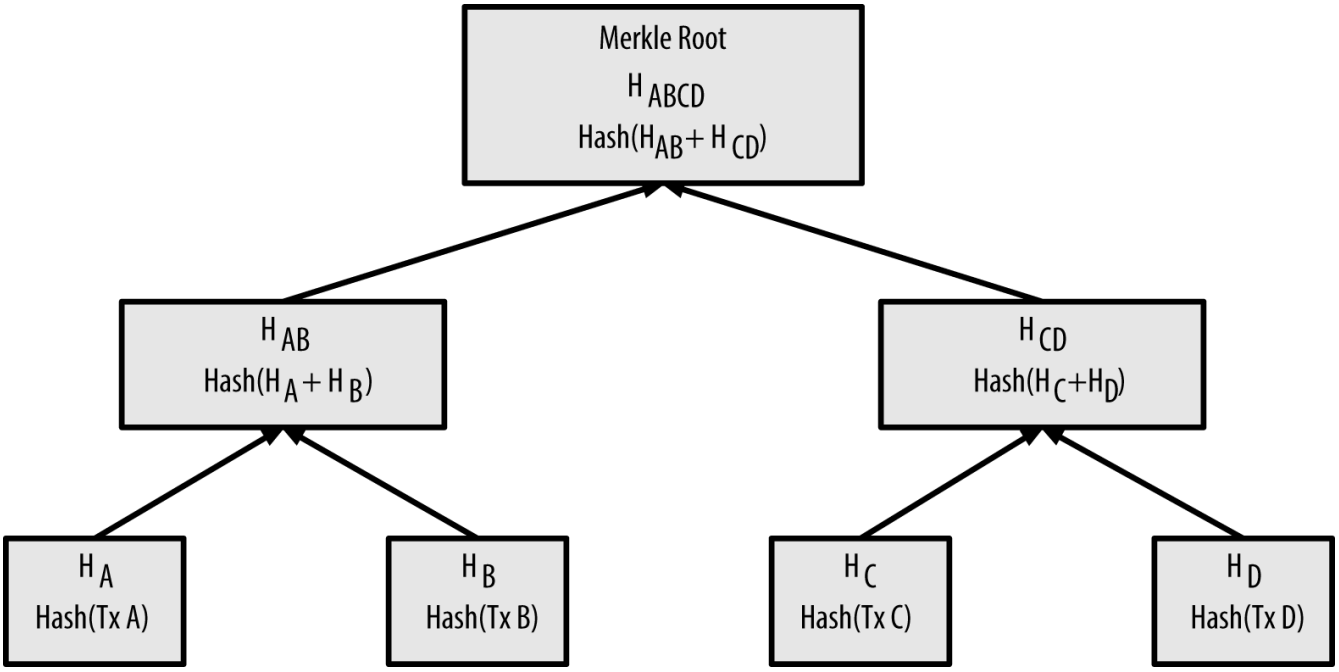

Figure 2. Calcolando i nodi in un merkle tree

Visto che il merkle tree è un albero binario, necessita di un numero pari di nodi foglia. Se il numero di transazioni da sintetizzare è un numero pari, l'ultimo hash di transazione sarà duplicato per creare un numero pari di nodi foglia, questo tipo di alberi sono conosciuti come *alberi bilanciati*. Il tutto è mostrato in Duplicando un elemento (data element) ottiene un numero di elementi pari, dove la transazione C è duplicata.

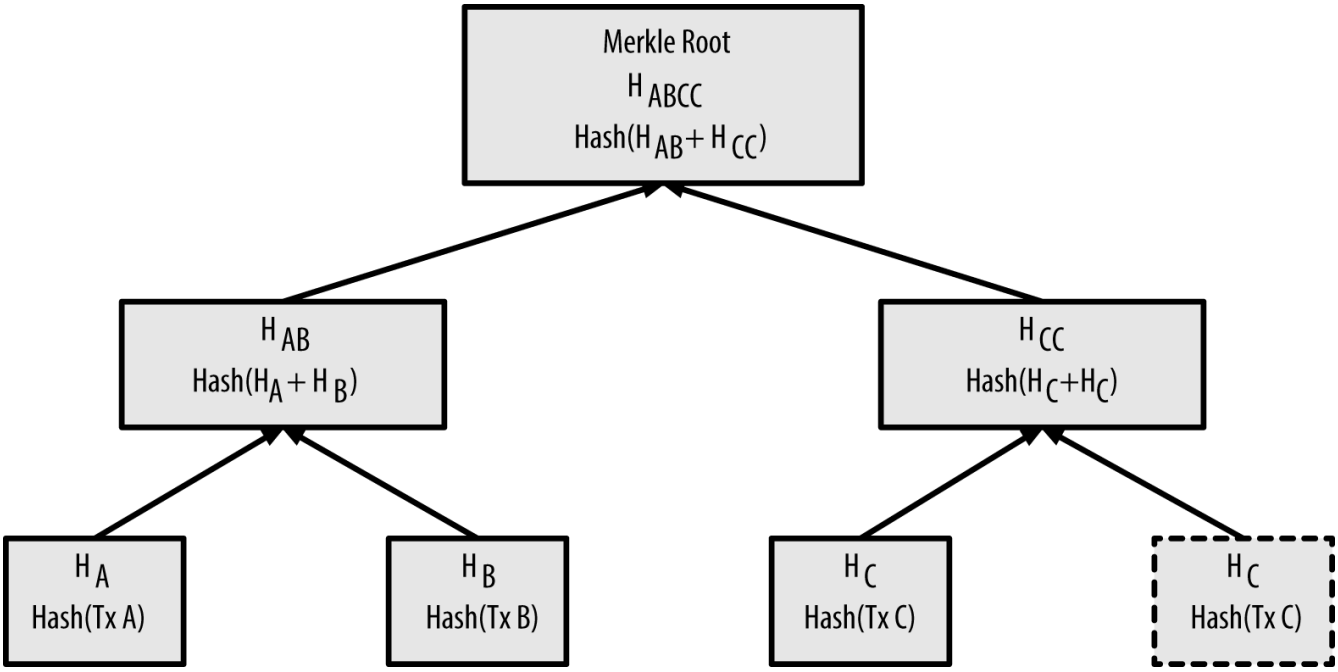

Figure 3. Duplicando un elemento (data element) ottiene un numero di elementi pari

Lo stesso metodo per costruire un tree da quattro transazioni può essere generalizzato per costruire trees di ogni dimensione. In bitcoin è comune avere diverse centinai di migliaia transazioni in un singolo blocco, che sarà indicizzato nel medesimo modo, producendo i 32 bytes di dati come un singolo merkle root. Nel Un merkle tree che riassume tanti data element, possiamo vedere un tree costituito da 16 transazioni. Nota che sebben il root sembra più grande rispetto ai nodi nel diagramma, esso è esattamente della stessa dimensione, cioè 32 bytes. Se c'è una transazione o un centinaio di migliaia di transazioni nel blocco, il merkle root li indicizza sempre in 32 bytes.

Per provare che una specifica transazione è inclusa nel blocco, un nodo ha bisogno solo di produrre degli hash di 32-byte log~2~(N) , costituendo un _percorso di autenticazione_o_percorso merke_connettendo la specifica transazione alla radice del tree. Ciò è particolarmente importante in quanto aumenta il numero di transazioni, poiché il logaritmo in base 2 del numero di transazioni aumenta molto più lentamente. Questo consente ai nodi bitcoin di produrre efficientemente il percorso di 10 o 12 hashes (320-384bytes),i quali possono provvedere alla prova della singola transazione da più di un migliaio di transazioni in un blocco di megabyte-size.

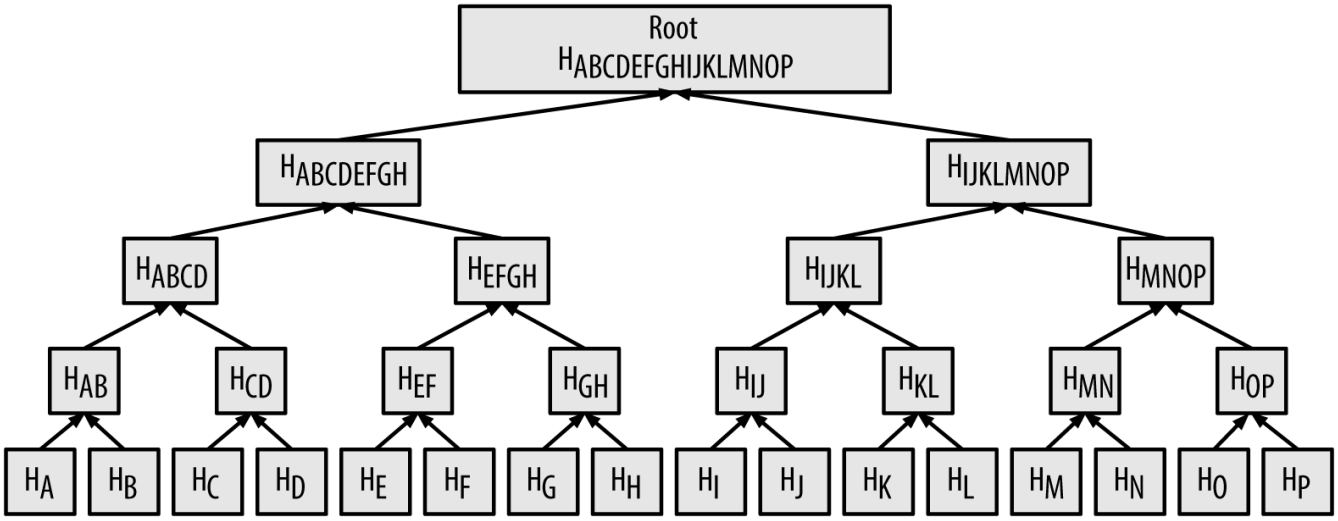

Figure 4. Un merkle tree che riassume tanti data element

Nel Un merkle path usato per provare l'inclusione di un data element, un nodo può dimostrare che una transazione K è inclusa nel blocco con la produzione di un percorso Merkle che è solo 32 byte di lunghezza dell'hash (128 byte totali). Il percorso si compone di quattro hash (indicato in blu in Un merkle path usato per provare l'inclusione di un data element) H_L, H_{IJ}, H_{MNOP} and $H_{ABCDEFGH}$. Con questi quattro hash forniti come un percorso di autenticazione, ogni nodo può dimostrare che H ~ K ~ (guarda il verde nel grafico) è incluso nella radice Merkle calcolando quattro ulteriori pair-wise hash H_{KL}, H_{IJKL}, $H_{IJKLMNOP}$ e la radice del Merkle tree root (illustrato in linea tratteggiata nella figura).

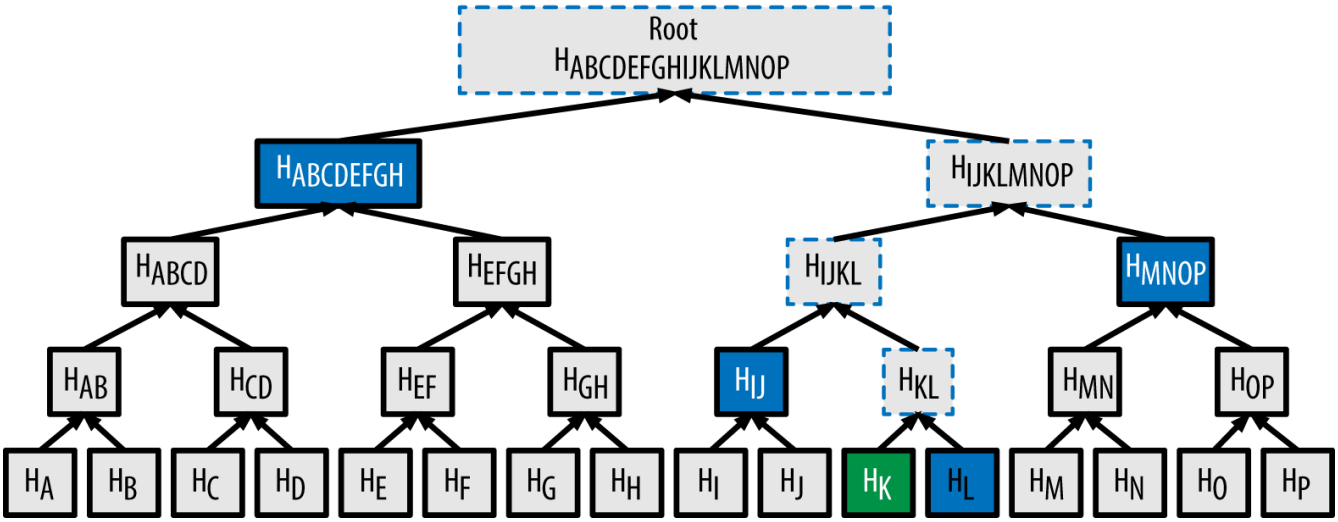

Figure 5. Un merkle path usato per provare l'inclusione di un data element

Il codice nel Costruendo un merkle tree dimostra il processo di creazione di un merkle tree dagli hash del nodo-foglia fino alla radice, usando la libreria libbitcoin per qualche funzione helper.

Example 1. Costruendo un merkle tree

```
#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_digest_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
        return bc::null_hash;
```

```
    else if (merkle.size() == 1)
        return merkle[0];

    // While there is more than 1 hash in the list, keep looping...
    while (merkle.size() > 1)
    {
        // If number of hashes is odd, duplicate last hash in the list.
        if (merkle.size() % 2 != 0)
            merkle.push_back(merkle.back());
        // List size is now even.
        assert(merkle.size() % 2 == 0);

        // New hash list.
        bc::hash_digest_list new_merkle;
        // Loop through hashes 2 at a time.
        for (auto it = merkle.begin(); it != merkle.end(); it += 2)
        {
            // Join both current hashes together (concatenate).
            bc::data_chunk concat_data(bc::hash_size * 2);
            auto concat = bc::make_serializer(concat_data.begin());
            concat.write_hash(*it);
            concat.write_hash(*(it + 1));
            assert(concat.iterator() == concat_data.end());
            // Hash both of the hashes.
            bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
            // Add this to the new list.
            new_merkle.push_back(new_root);
        }
        // This is the new list.
        merkle = new_merkle;

        // DEBUG output -------------------------------------
        std::cout << "Current merkle hash list:" << std::endl;
        for (const auto& hash: merkle)
            std::cout << "  " << bc::encode_hex(hash) << std::endl;
        std::cout << std::endl;
        // --------------------------------------------------
    }
    // Finally we end up with a single item.
    return merkle[0];
}

int main()
{
    // Replace these hashes with ones from a block to reproduce the same merkle
root.
    bc::hash_digest_list tx_hashes{{

bc::decode_hash("0000000000000000000000000000000000000000000000000000000000000000"
),
```

```
bc::decode_hash("0000000000000000000000000000000000000000000000000000000000000011"
),

bc::decode_hash("0000000000000000000000000000000000000000000000000000000000000022"
),
    }};
    const bc::hash_digest merkle_root = create_merkle(tx_hashes);
    std::cout << "Result: " << bc::encode_hex(merkle_root) << std::endl;
    return 0;
}
```

Compilando e eseguendo il codice di esempio dell'esempio merkle mostra il risultato della compilazione e esecuzione del codice merkle.

Example 2. Compilando e eseguendo il codice di esempio dell'esempio merkle

```
$ # Compila il codice in merkle.cpp
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Esegui l'eseguibile merkle
$ ./merkle
Lista di hash merkle corrente:
  32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
  30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Lista di hash merkle corrente:
  d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Risultato: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3
```

L'efficenza dei merkle tree diventa ovvia man mano che la scala aumenta. Efficenza del merkle tree mostra la quantità di dati necessari che devono essere scambiati come merkle path per provare che la transazione sia facente parte di un blocco.

Table 3. Efficenza del merkle tree

Numero di transazioni	Dimensione approssimativa del blocco	Dimensione del path (hash)	Dimensione del path (byte)
16 transazioni	4 kilobyte	4 hash	128 byte
512 transazioni	128 kilobyte	9 hash	288 byte
2048 transazioni	512 kilobyte	11 hash	352 byte
65,535 transazioni	16 megabyte	16 hash	512 byte

Come si può vedere dalla tabella, mentre la dimensione del blocco aumenta rapidamente, da 4 KB con 16 transazioni a una dimensione di blocco di 16 MB per adattarsi 65.535 transazioni, il percorso Merkle necessario per provare l'inserimento della transazione aumenta molto più lentamente, da

128 bytes a soli 512bytes. Con il merke trees, un nodo può scaricare soli i block headers (80 bytes per blocco) e può essere ancora in grado di identificare l'inserimento della transazione nel blocco recuperando una parte del percorso merkle da un full node, senza memorizzare o trasmettere la stragrande maggioranza della blockchain, che potrebbe essere di diversi gigabyte di dimensione. I nodi che non contengono una blockchain intera, chiamati simplified payment verification (SPV nodes), usano il merkle path per verificare le transazioni senza scaricare gli interi blocchi.

Merkle Tree e Simplified Payment Verification (SPV)

I merkle tree sono usati estensivamente dai nodi SPV (o VSP in italiano - Verifica Semplificata del Pagamento, ndt). I nodi SPV non hanno tutte le transazioni e non scaricano blocchi completi, ma solo header dei blocchi. Per verificare che una transazione sia inclusa in un blocco, senza dover scaricare tutte le transazioni nel blocco, essi utilizzano un percorso di autenticazione, chiamato anche merkle path.

Consideriamo, per esempio, un nodo SPV che è interessato ai pagamenti in entrata all'indirizzo contenuto nel suo wallet. Il nodo SPV stabilirà un filtro bloom sulle sue connessioni ai peer per limitare le transazioni ricevute ai soli che contengono indirizzi di interesse. Quando un peer vede una transazione che corrisponde al filtro bloom, invierà quel blocco utilizzando un merkleblock messaggio. Il merkleblock messaggio che contiene il block header, così come un percorso Merkle che collega la transazione di interesse alla radice Merkle nel blocco. Il nodo SPV può utilizzare questo percorso Merkle per collegare la transazione al blocco e verificare che l'operazione è inclusa nel blocco. Il nodo SPV utilizza anche il block header per collegare il blocco al resto della blockchain. La combinazione di questi due link, tra la transazione e il blocco, e tra il blocco e blockchain, dimostra che l'operazione è registrata nella blockchain. Tutto sommato, il nodo SPV avrà ricevuto meno di un kilobyte di dati per l'header di blocco e il merkle path, una quantità di dati che è più di mille volte inferiore a un blocco completo (circa 1 megabyte al momento).